揭秘 jbpm 流程引擎内核设计思想及构架

1 前言

流程引擎内核仅是“满足 Process 基本运行”的最微小结构，而整个引擎则要复杂很多，包括“状态存储”、“事件处理”、“组织适配”、“时间调度”、“消息服务”等等外围的服务性功能。引擎内核，仅包含最基本的对象和服务，以及用于解决流程运行问题的调度机制和执行机制。

如果，你掌握了一个流程引擎的灵魂，你才有能力理解它的全部。否则，一个引擎对你来说，可能只是一个复杂的结构， 丰富多彩 API 、令人眼花缭乱的 “功能”和“服务”而已。

本身工作流这个领域就是一个很 “狭窄”的领域，国内的厂商也不是很多， 其中有部分实现技术并不弱。但可能涉于安全等因素， 并没有多少技术人员探讨 “深度的工作流技术实现问题”。而广大的开发爱好者却还在花费大量的时间在摸索 “如何理解工作流、如何应用工作流”。所以在此之前，国内尚未有一篇技术文章探讨工作流引擎内核的实现，当然也没有探讨 jBpm 引擎内核的文章了。在 www.javaeye.com 技术站点和我的 blog （h

ttp://blog.csdn.net/james999 ）上有几篇专门探讨 jbpm 应用的文章，对于初步想了解如何使用 jbpm 的读者来说，值得看看。

对于这方面的技术分享，开源是个不错的突破口。

本篇就是以 jBpm 为实例，来诠释工作流引擎的内核设计思路和结构。但是这仅仅是从 jBpm 的实现角度来辅助大家理解， 因为工作流引擎内核的设计、 实现是有很多方式：这会因所选的模型、调度算法、推进机制、状态变迁机制、执行机制等多方面的不一样，而会差别很大。 比如基于 Activity Diagram 模型的 jBpm 和基于 FSM 模型的 OSWorkflow 引擎内核之间就有很大的差别。

相比较而言， jBpm 的模型比较复杂， 而引擎内核实现的比较 “精简 ”，非常便于大家“由浅入深的理解 ”。

2 阅读本篇的基础准备

2.1 概念的基础

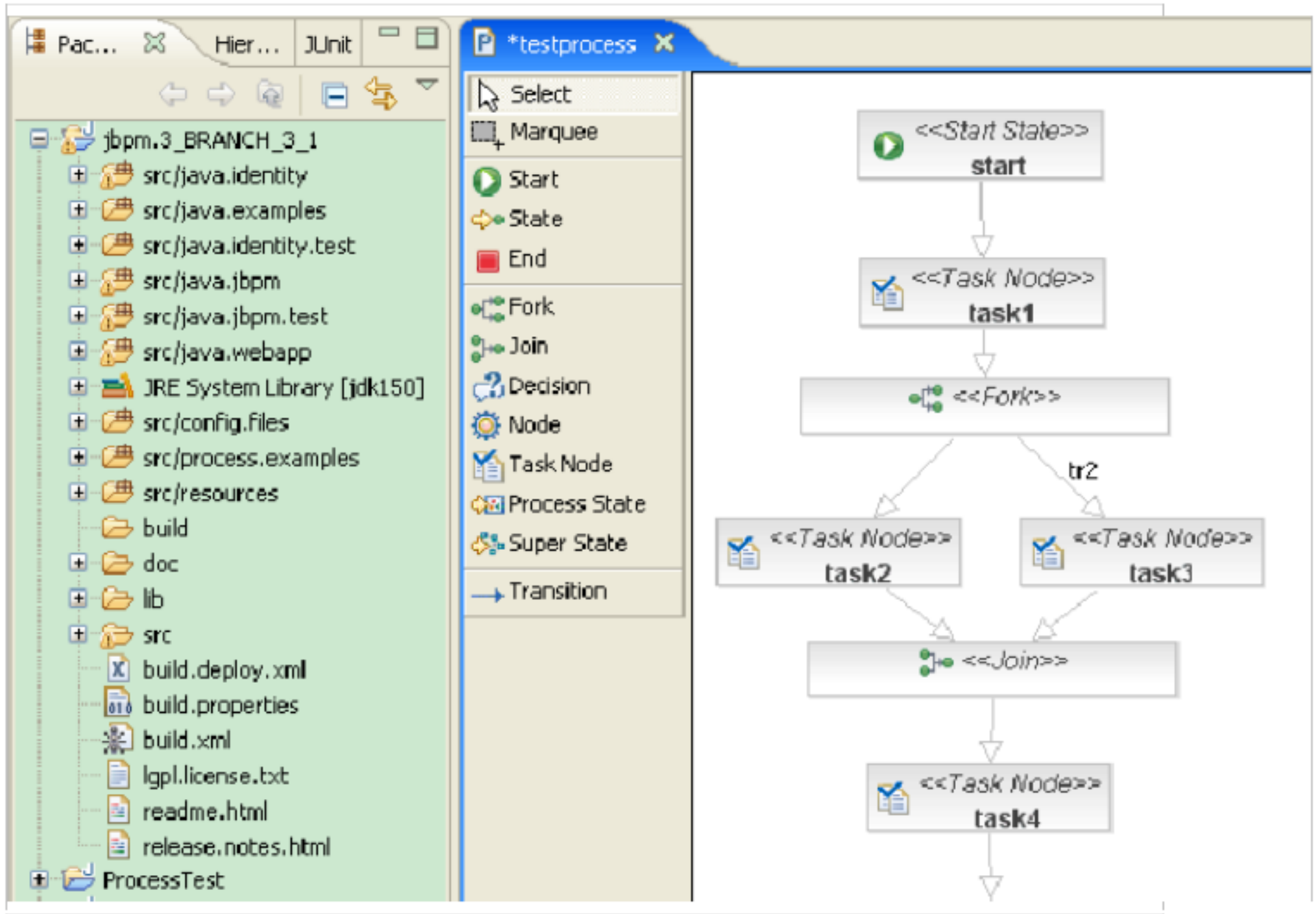
本文的读者群主要是面向有一定工作流基本概念的开发人员。 所以本文认为你已经具备了如下基本工作流知识：

（1） 初步了解工作流系统结构。比如理解工作流引擎在工作流系统中所处的位置和作用

（2） 对流程定义（ Process Definition ）和流程实例（ Process Instance ）相关对象有所了解。比如理解 Process Instance 代表什么，工作项（ WorkItem ）代表什么。

2.2 环境的基础

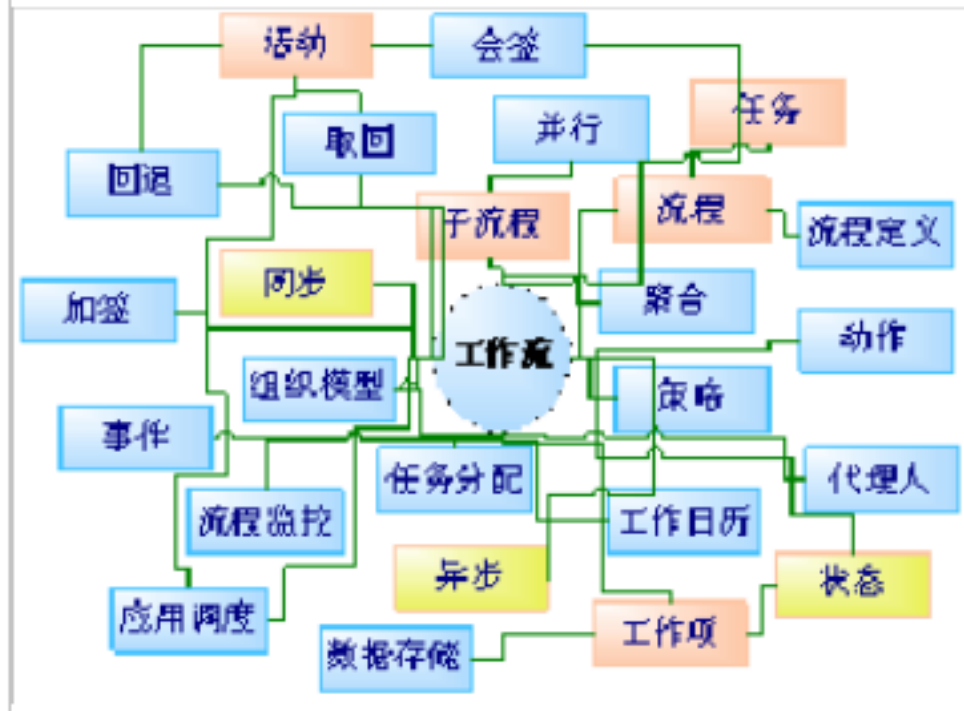
在阅读本篇的时候， 如果你已经搭建了一套 jbpm 的开发环境， 那么将有助于你更容易理解本篇的很多内容，也便于实际体验代码。从 www.jbpm.org 官方网站下载 jbpm-starters-kit 开发包， 按照其参考手册， 可以很容易在 eclipse 开发环境中建立项目，效果图类似如下：

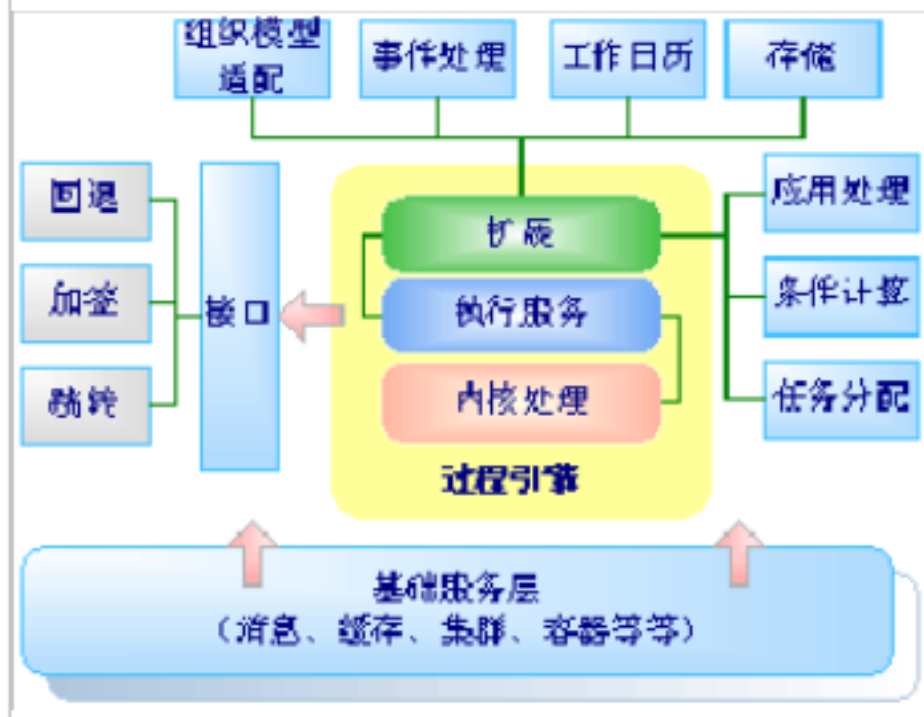


3　什么是流程引擎内核？

我比较推崇 “微内核的流程引擎构架 ”, 并在最近两三年内写了两篇探讨此方面的文章：第一篇是写于 05 年 7 月份的《微内核流程引擎架构体系》 ，第二篇是 07 年 7 月份的《微内核过程引擎的设计思路和构架》（受普元《银弹》杂志约稿所写，尚未对外公开）。

但至今对外阐述引擎内核到底是什么。





正如上面的两张图所示，我们可以通过 “微内核 ”的构架来使得流程引擎的结构更加 “清晰 ”。而能否实现 “微内核 ”的根本，则是看你是否能够设计并抽象出 “良好的引擎内核结构 ”。

很显然， 要想设计出一套结构优良的引擎内核， 首要条件就是： 明白什么是引擎内核。

首先我们需要明白引擎是什么，引擎可以做什么。这在 WfMC 的《工作流参考模型》中已经有很详细的解答，本文不再重复。知道这个仅仅是不够的，你还需要很清晰的明白如何去 “为流程建模 ”，而这则在 Aalst 大师所著的《工作流管理 —— 模型、方法、系统》一书有细致阐述，本文也不再重复。

但很可惜， 至今尚未有一本专门的书籍来论述 “过程建模方法 ”的，或者说如何利用这些既有的 “过程建模方法（诸如 FSM 、PetriNet 、 EPC 、 Activity Diagram 等等） ”来解决流程问题。这个只能分别查阅相关资料，此处也不叙述。

因为文本只讲 “引擎内核 ”。

如果我们暂且把那复杂的流程业务性问题，诸如 “组织模型分配 ”、 “分支条件计算 ”、 “事件处理 ”、“消息调度 ”、 “工作项处理 ”、“存储 ”、 “应用处理 ”、以及那些 “变态的诸如会签、回退之类的模型 ”都统统的抛弃， 只留下 “最单纯的过程性问题 ”，也就是 “解决一个

过程运行问题，　按秩序的从一个节点到另一个节点的执行　　”。—— 这就是引擎内核所关注的根本问题。

上面这句话，　估计会引起很多人　“拍砖 ”。在很多人看来，　工作流之所以看起来很　　“难 ”，就是因为这些复杂多变的　　“业务性问题 ”都统统绑在一个　“引擎 ”上造成的。

其实，这是两个　“维度 ”的问题，也就是　“引擎的抽象 ”和 “引擎的应用 ”这两个不同维度，不同层面的问题。　但这绝不是两个独立的问题，　　“引擎的抽象 ”的好与坏，　直接影响到　“引擎的应用 ”的可复杂度和可支持度，　　当然我们也不能否认，　　“引擎的应用 ”问题也是一个很复杂的问题。但本文是站在　　“引擎的抽象 ”这个维度来阐述问题的。对于　　“引擎的应用 ”问题，可参考我的前作：　2003　年 11 月份的《工作流模型分析》、　　2003　年 12 月份的《工作流授权控制模型》、　2004　年 7 月份的《工作流系统中组织模型应用解决方案》。

也就是说，　本文不是指导大家如何去　　“使用 jbpm ”，而是阐述 “ jbpm 的引擎的内核部分是如何构建的　”。但本文的主旨不是告诉大家　　“ jBpm 是如何设计引擎内核的　　”，而是以 jBpm　为例，来介绍　“引擎内核 ”。

4　引擎内核所关注的四个主要问题

引擎内核所关注的是一个非常　　“抽象 ”层面的问题，而不同引擎关注的　　“一套完整的执行环境 ”。或者我们可以这么来说，　　引擎内核的职责是非常　　“精简 ”的：确保流程按照既有的定义，从一个节点运行到另一个节点，并正确执行当前节点。

总的来说，引擎内核主要关注四个方面的问题：

（1）　流程定义问题：不是说如何图形化的定义流程，而是如何用一套定义对象，来诠释所定义的流程。

（2）　流程调度问题：　提供什么的机制，　可以确保流程能够处理复杂的　　“流程图结构 ”，诸如串行、并行、分支、聚合等等，并在这复杂结构中确保流程从一个节点运行到另一个节点。

（3）　流程执行问题：当流程运行到某个节点的时候，需要一套机制来解决：是否执行此节点，并如何执行此节点的问题，并维持节点状态生命周期。

（4）　流程实例对象：需要一整套流程实例对象来描述流程实例运行的状态和结果。

4.1　模型与定义对象

工作流引擎本身就是一种　“ base on model ”的组件，流程实例的执行都是依赖于所定义的 “流程定义 ”，而工作流引擎则是提供了这样一种环境，来维持流程实例的运行。

所以引擎内核，必须提供一套定义对象来描述“流程定义”，并且这些定义对象必须反映出一种“模型”。

比如 jBpm 的定义对象，是与其所基于的 Activity Diagram 模型相对应的。

4.2 调度机制与算法

引擎内核的另一个重要功能，就是保证流程实例准确的从一个节点运行到另一个节点，而这则需要依赖于一套调度机制。

引擎的调度机制有很多种实现方法，有的甚至是与“所依赖的模型有关”。但普遍来讲，很多引擎都受到 Petri Net 的影响，而采用 token 来调度。

jBpm 本身就吸纳的 token 这套机制，当然，与 Petri Net 的调度机制还是有所区别。我们将在下面的章节详细介绍。

4.3 执行机制与状态

经过引擎的调度，实例运行到某个节点了，此时必须必须提供一套机制，来判断当前节点是否可执行，如果可执行，那么需要提供一套 runtime envrioment 来执行节点——这就是引擎的执行机制。

复杂的流程引擎会依赖于“流程实例状态”或“活动实例状态”的约束和变迁来进行处理。之所有有时候我们会把一个流程引擎也叫做“状态机”，很大程度上也是这个原因。

4.4 实例对象与执行环境

每个一个流程实例，必须维护一套属于自己的“运行环境和数据”，而这则是实例对象的责任了。基本上实例对象会包含如下信息：

（1） 与流程实例的状态或控制信息

（2） 与活动实例的状态或控制信息。如果某些引擎不支持活动实例，那么必然会有某些其他实例信息，可以当前节点的状或控制信息。

（3） 一些临时的“执行”信息，便于引擎针对某种情况进行处理

5 jbpm，“精简”的开源流程引擎

好的开源工作流引擎不多，jbpm 和 osworkflow 算是其中两个有特色而且比较容易实际应用的。目前一些国内的中小型流程应用项目，就是在 jbpm 或 osworkflow 的基础上扩展实现。jBpm 采用了 Activity Diagram 的模型，而 osworkflow 则是 FSM 的模型。

当然，这仅仅是 jbpm3 之后的事情。自从被 Jboss 收购之后， jbpm 对早先的 2.0 构架进行了重组，整个结构完全本着 “微内核 ”的思想进行设计。

现在这里从技术角度来分析 jbpm3 的优点，简单罗列几个大家都容易看见的：

（1） jbpm 的模型是采用 UML Activity Diagram 的语义，所以便于开发人员理解流程。

（2） jbpm 提供了可扩展的 Event-Action 机制，来辅助活动的扩展处理。

（3） jbpm 提供了灵活的条件表达式机制，来辅助条件解析、脚本计算的处理。

（4） jbpm 提供了可扩展的 Task 及分配机制，来满足复杂人工活动的处理。

（5） 借助 hibernate 的 ORM 的优势， jbpm 能够很容易支持多种数据库。

当然，还有一些优点，是很多开发人员并不太注意的，比如：

（1） jbpm 的 Node 机制非常灵活，开发人员可以很容易定制 “业务化语义的节点 ”，并满足运行时候处理的需要。

有很多灵活的优点，当然也少不了存在一些 “局限 ”。

（1） 很显然，只能有一个 start-state 。

（2） jbpm 依靠 Token 来调度和计算，在同一个时刻中，一个 ProcessInstance 只允许一个 Token 对象只存在一个 Node 中（分支当然用 Child Token 对象处理）。所以本质上就不支持 “ multi - instance ” 模式。

（3） jbpm 作为一款开源的工作流引擎，其更多的是关注 “如何辅助你更容易的让流程运行完成 ”，但是并不记录 “流程运行的历史和轨迹 ”。这一点可能是东西方文化的差异性所在， 因为国内的流程应用，比较关注 “运行轨迹 ”。

至于其他的一些局限，比如不支持 “回退 ”、 “跳转 ”等操作，这也是因为东西方文化的差异所在。 西方人认为 “往回流转的情况肯定也是一种业务规则所定义， 那么肯定可以通过分支或条件来解决 ”，而东方则把 “回退作为一个人性化管理和处理的潜在特点 ”。所以诸如此类的一些 “特定需求 ”，估计只能通过扩展 jbpm 来实现了， 甚至有时候， 简单的扩展是无法解决问题的 —— 正如上一节所说的那样， “引擎的抽象 ”会影响 “引擎的应用 ”的复杂度支持。

但是，当你试图修改 jbpm 代码的时候，你会顾虑 jbpm 的 LGPL 协议吗？（很多国内企业从来不考虑这个协议问题，寒）。

6 jBpm 流程模型与定义对象

6.1 首先解决如何形式化描述一个流程的问题

这里说的 “定义流程 ”并不是说 jbpm3 中那个基于 eclipse plugin 的图形化建模工具。而是 如何去解决 “形式化的描述一个流程 ”的问题。

形式化的描述流程并不是一个简单的问题，从上世纪七十开始，人们就在探索用各种各样多的模型来描绘流程： Petri Net, FSM, EPC, Activity Diagram, 以及近来的 XPDL MetaModel 等等，延伸到如今的 BPEL,BPMN,BPMD 等等。

jBpm 采用了 Activity Diagram 的模型语义：其将用 Start State 、State 、Action State （Task Node ）、 End State 、Fork 、Join 、Decision 、Merge 、Process State 这几个 “元素 ”的组合来描述任何一个流程。其中 Action State 是 Activity Diagram 中的标准语义，在 jBpm 为了便于大家理解和使用， jBpm 采用了 TaskNode 这个语义。

在 WfMC 的 Workflow Reference Model 中，对流程引擎的功能描述， 其中就包含一项：解析流程定义。如果想满足这这功能，前提条件就必须有最基本的两个：

（1） 有一套形式化的描述语言（通常为 xml 格式）。利用这个描述语言可以描述一个流程的定义。比如 WfMC 所提出的 XPDL 这个描述语言。当然， jBpm 也有自己的一套，名为 jPDL ，也是一个 xml 格式的。

（2） 有一套对象集可以反映流程的定义模型和结果， 一般叫做定义对象。 流程引擎就需要把 “ xml 格式的流程定义 ”解析为一套对象，而这套对象的结构则反映了流程的结构。

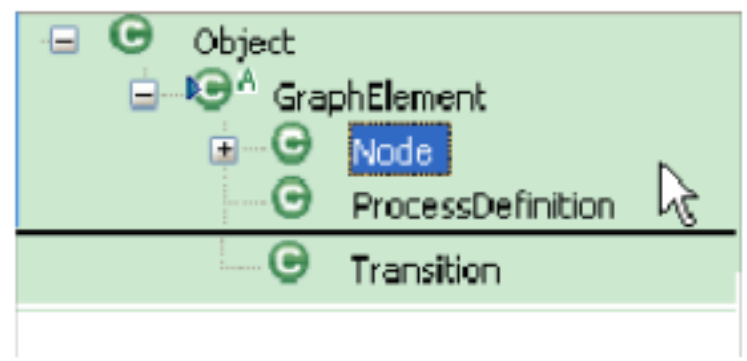
我们暂且不去探讨 jPDL 那个形式化的 xml 语言，而把重心放在 jBpm 那套定义对象中。因为这个定义对象是属于 Engine Kernel 的一部分。

6.2 抽象的节点（ Node ）和转移（ Transition ）

面向对象的继承性、多态性可以让我们从最抽象的部分来描述对象。那么这套定义对象也需要从最基础的 “抽象 ”说起。

process 的本质就是 “节点 ”和 “有向弧 ”，当然你也可以说是 Node 和 Link ，或者 Node 和 Transition ，或者 Activity 和 Transition 等等之类的。 jBpm 采用的是 Node 和 Transition 来表示 “节点 ”和 “有向弧 ”。

于是乎，在 jbpm 中你可以看到这样的结构关系：



对于一个节点来说，从定义角度，其只关心几个事情：

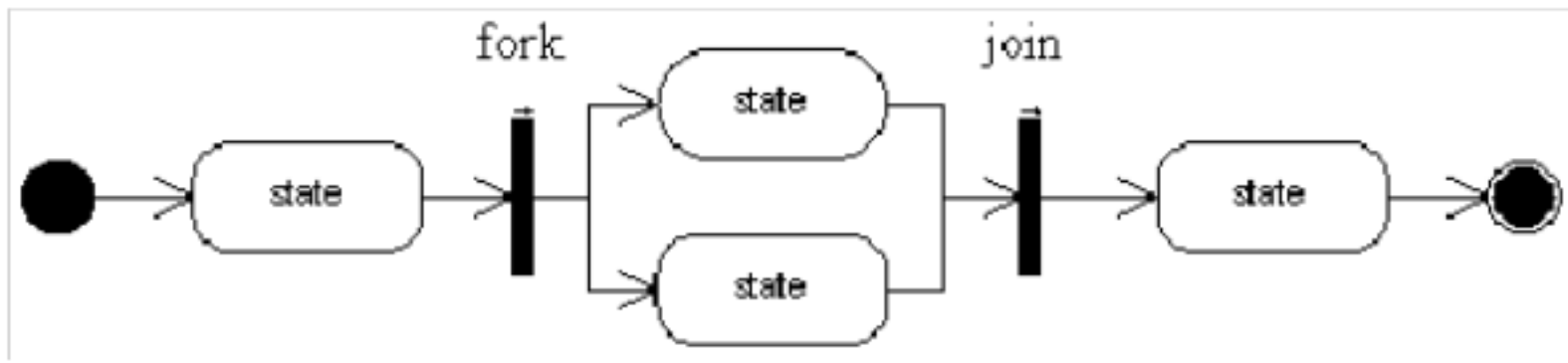
（1） 这是个什么类型的节点。这个节点可能是 start state ，也可能是一个 task node ，或者是一个 fork 。

（2） 这个节点的转入 Transition 和转出 Transition 。

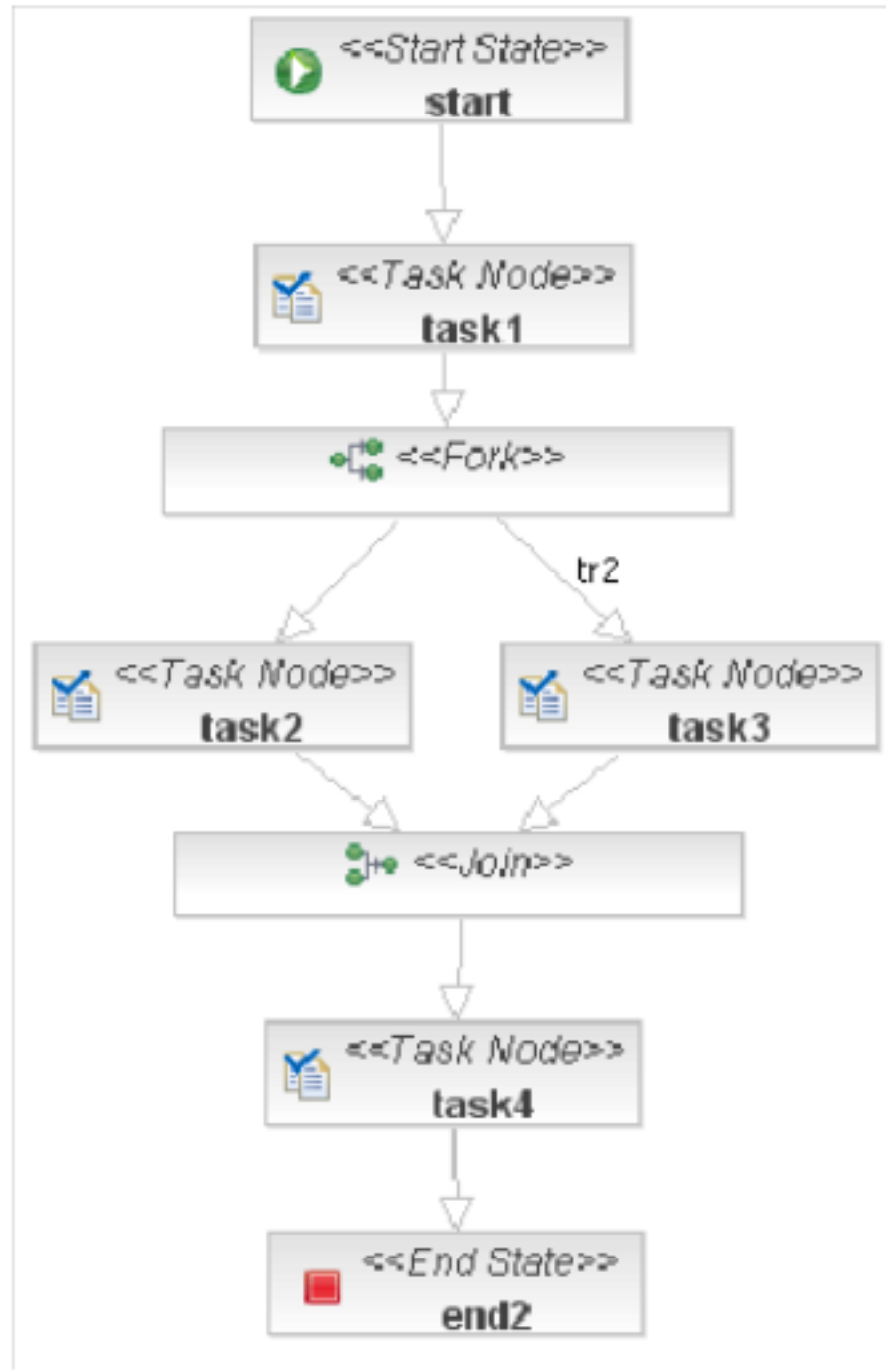
可能有的人会说，还需要关心节点的转入转出的类型，比如 And Splite 或者 Xor Join 之类。这个并没有错，因为很多流程模型的节点元素需要考虑这个，比如 WfMC 的 XPDL 模型。但是 jBpm 的节点是没有这样的属性的，或者说的更准确些，是 Activity Diagram 模型的节点没有这样的特性。活动图是采用 “ Fork 、” “ Join 这样的节点来解决 “分支 ”问题。

6.3 流程：节点与转移的组合

仅利用节点和转移的组合，就可以表达一个 “过程（ Process ） ”。当然这个流程只能告诉人们 “大概的业务过程 ”，当然不包括很复杂的信息。如下图所示：



这是一张非常标准的 “活动图 ”, 如果我们用 jbpm 的设计器，看看这样一张 “流程图 ”：



不论你如何绘画，改变不了这张图的本质：它就只有两个基本元素：节点和转移。只是有的节点是 start-state ，有的是 task-node ，有的是 join ，有的是 end state 而已。

6.4 节点的类型和扩展

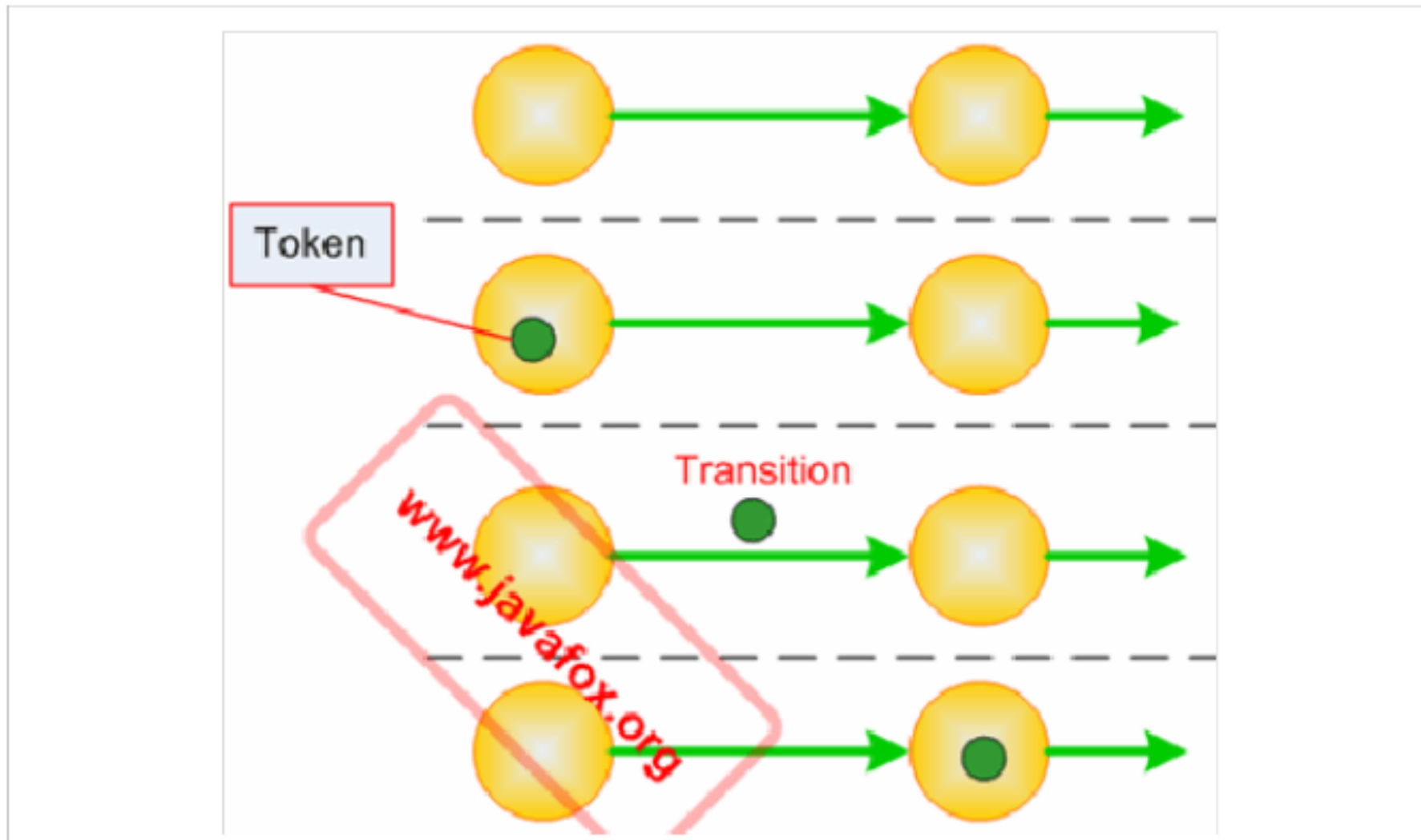
我们可以通过定义自己的 Node 节点对象，来补充 jbpm 自定的节点对象。只需要 extends Node ，并重写读写 xml 的 read 和 write 方法，重写负责执行的 execute 方法，在 org/jbpm/graph/node/node.types.xml 中配置即可，当然，你可以写的更加复杂，更加业务化的节点。

7 jBpm 的过程调度机制

7.1 吸纳自 Petri Net 思想

jBpm 的过程调度机制是吸纳了 Petri Net 的一些思想。

jBpm 采用 Token 来表示当前实例运行的位置，也利用 token 在流程各个点之间的转移来表示流程的推进，如下图所示：



当 jbpm 试图去启动一个流程的时候，首先是构造一个流程实例，并为此流程实例创建一个 Root Token ，并把这个 Root Token 放置在 Start Node 上。

以下截取部分代码实现，仅供参考。手头有 jbpm3 相应开发环境的朋友，可以打开 ProcessInstance 和 Token 这两个类。（注：以下所有参考代码，为了突出主题，都已经将实际代码中的 event,log 等处理删除）

```
public ProcessInstance( ProcessDefinition processDefinition ) {
        this.processDefinition = processDefinition;
        this.rootToken = new Token(this);
```

```
public Token(ProcessInstance processInstance) {
        this.processInstance = processInstance;
    this.node = processInstance.getProcessDefinition().getStartState();
```

jbpm 是允许在 start-state 执行 Task 的，也允许在 start-state 创建工人任务。不过此处我们不予讨论。

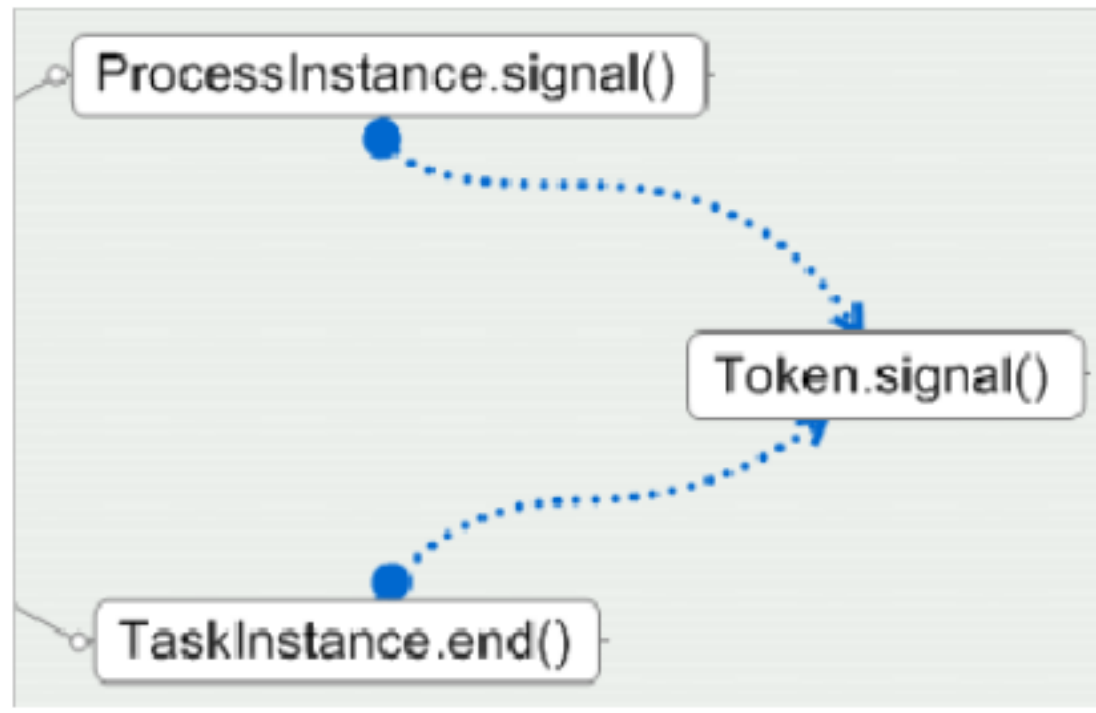
7.2 Token 的推进

当 Token 已经在 Start-State 节点了，我们可以开始往前推进，来促使流程实例往前运行。对于外部操作来说，触发流程实例往下运行的操作有两个：

（1） 强制执行 ProcessInstance 的 signal 操作

（2） 执行 TaskInstance 的 end 操作。

但是，这两个操作，都是通过 “当前 token 的 signal 操作 ”来内部实现的，如下图所示：



Token 的 Signal 操作表示： 实例需要离开当前 token 所在的节点， 转移到下一个节点上。 因为 Node 与 Node 之间是 “ Transition ” 这个桥梁， 所以， 在转移过程中， 会首先把 Token 放入相关连的 Transtion 对象中，再由 Transition 对象把 Token 交给下一个节点。

让我们来看看 Token 类中 signal 方法的部分代码实现，仅供参考：

```
public void signal() {
// 注意 ExecutionContext 对象
signal(node.getDefaultLeavingTransition(), new
ExecutionContext(this) );
}

void signal(Transition transition, ExecutionContext executionContext) {
// start calculating the next state
node.leave(executionContext, transition);
}
```

接下来，请注意 node.leave() 这个操作。这是一个很有意思的语义转换：我们是采用 token 的 signal 操作来表示往下一个节点推进，但是实际确实执行的 node.leave () 操作。

如果这地方让你自己来实现，代码会不会就是这样子呢？不妨此处想一想。

```
// 假设代码，仅供思考

void signal(Transition transition, ExecutionContext executionContext) {

transition.take(executionContext);

}
```

前面说过，jbpm 的调度机制吸纳的 Petri Net 的思想。在 Petri Net 中，并没有 transition 中驻留 token 这个语义，token 只驻留在库所（Place）中。所以，jbpm 此处的设计思路，是于此有一定关系的。所以只是把一个 ExecutionContext 对象放在了 transition 中，而不是一个 token 对象。

让我们来看看 node 对象的 leave 方法：

```
public void leave(ExecutionContext executionContext, Transition

transition) {

Token token = executionContext.getToken();

token.setNode(this);

executionContext.setTransition(transition);

executionContext.setTransitionSource(this);

transition.take(executionContext);

}
```

我们直接跟踪进 Transition 的 take 操作：

```
public void take(ExecutionContext executionContext) {

executionContext.getToken().setNode(null);

// pass the token to the destinationNode node

to.enter(executionContext);

}
```

经过这么多的中间步骤，我们终于把 ExecutionContext 对象从一个 node 转移到下一个 node 了。让我们来看看 Node 对象的 enter 操作：

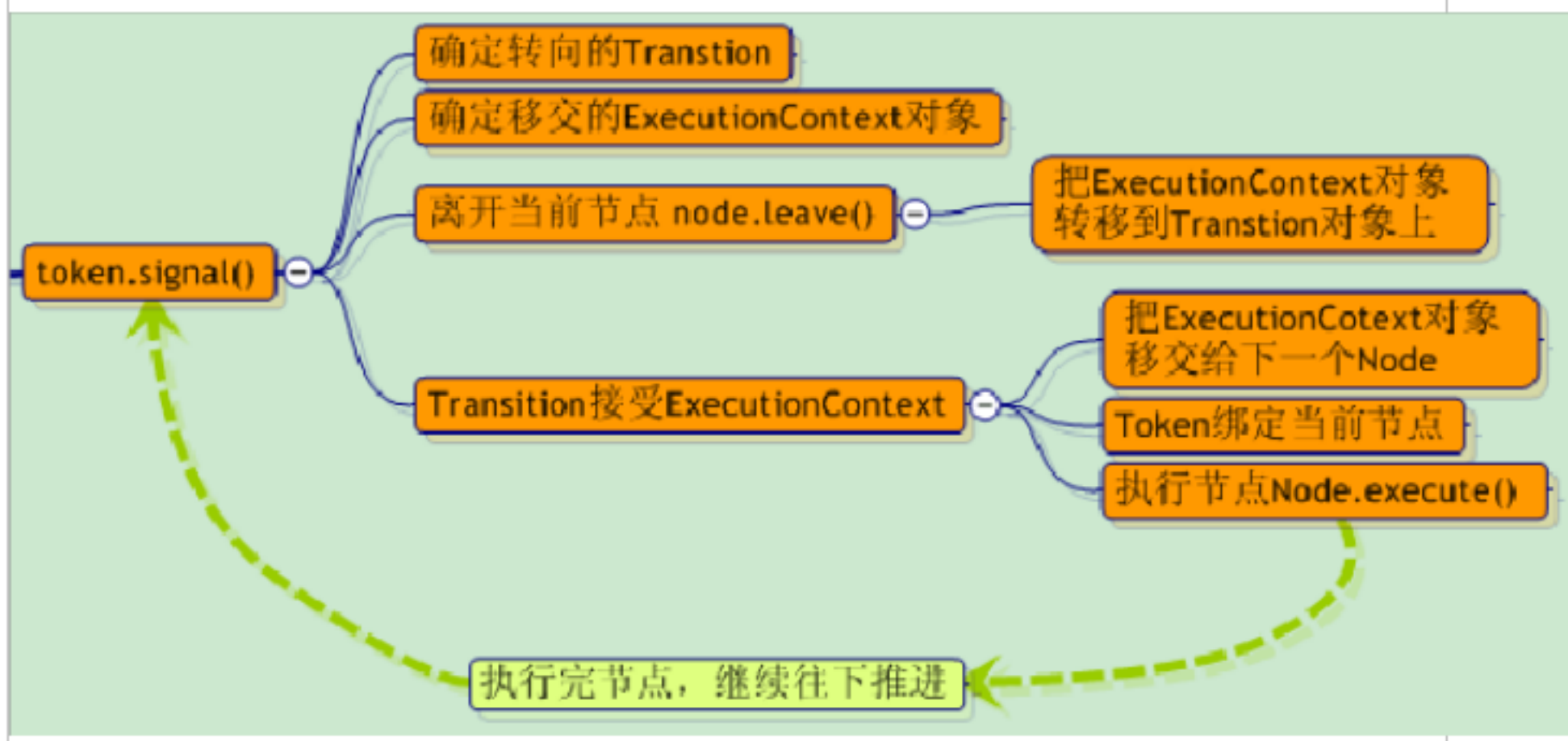
```
public void enter(ExecutionContext executionContext) {

Token token = executionContext.getToken();

token.setNode(this);

// remove the transition references from the runtime context

executionContext.setTransition(null);

executionContext.setTransitionSource(null);



// execute the node

if (isAsync) {



} else {

execute(executionContext);

}

}
```

至此， jBpm 成功的从一个节点转移到下一个节点了。 —— 这就是 jbpm 的调度机制。

7.3 非常简单的调度机制

怎么样，是不是非常的简单？

让我们把整个过程，用一张更清晰的 “思维图 ”来展示一下：



8 jBpm 的过程执行机制

8.1 执行机制

前面我们的“过程调度机制”是为了让流程可以正确的从“一个节点转移到下一个节点”，而本节所要讲解的 jbpm“执行机制”，则是为提供一个运行机制，来保证“节点的正确执行”。

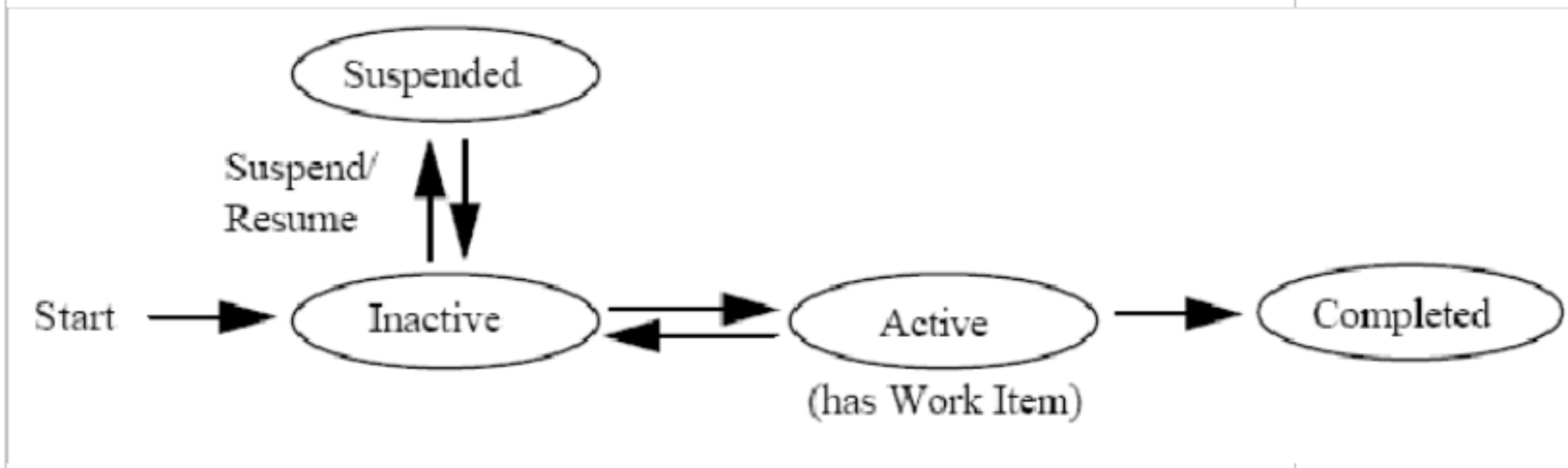
首先我们需要明确如下的概念：

（1） 节点有很多中，每种节点的执行方式肯定是不一样的

（2） 节点有自己的生命周期，不同的生命周期阶段，所处的状态不同。

在 WfMC 的《工作流参考模型》文档中，为活动实例归纳了几个可参考的生命周期。

（仅供参考，实际很多工作流引擎的节点的生命周期要比这复杂）



但是，jbpm 并没有突出“节点生命周期”这个理念，仅仅只是在“Event”中体现出出来。在我看来，可能的原因有两个：

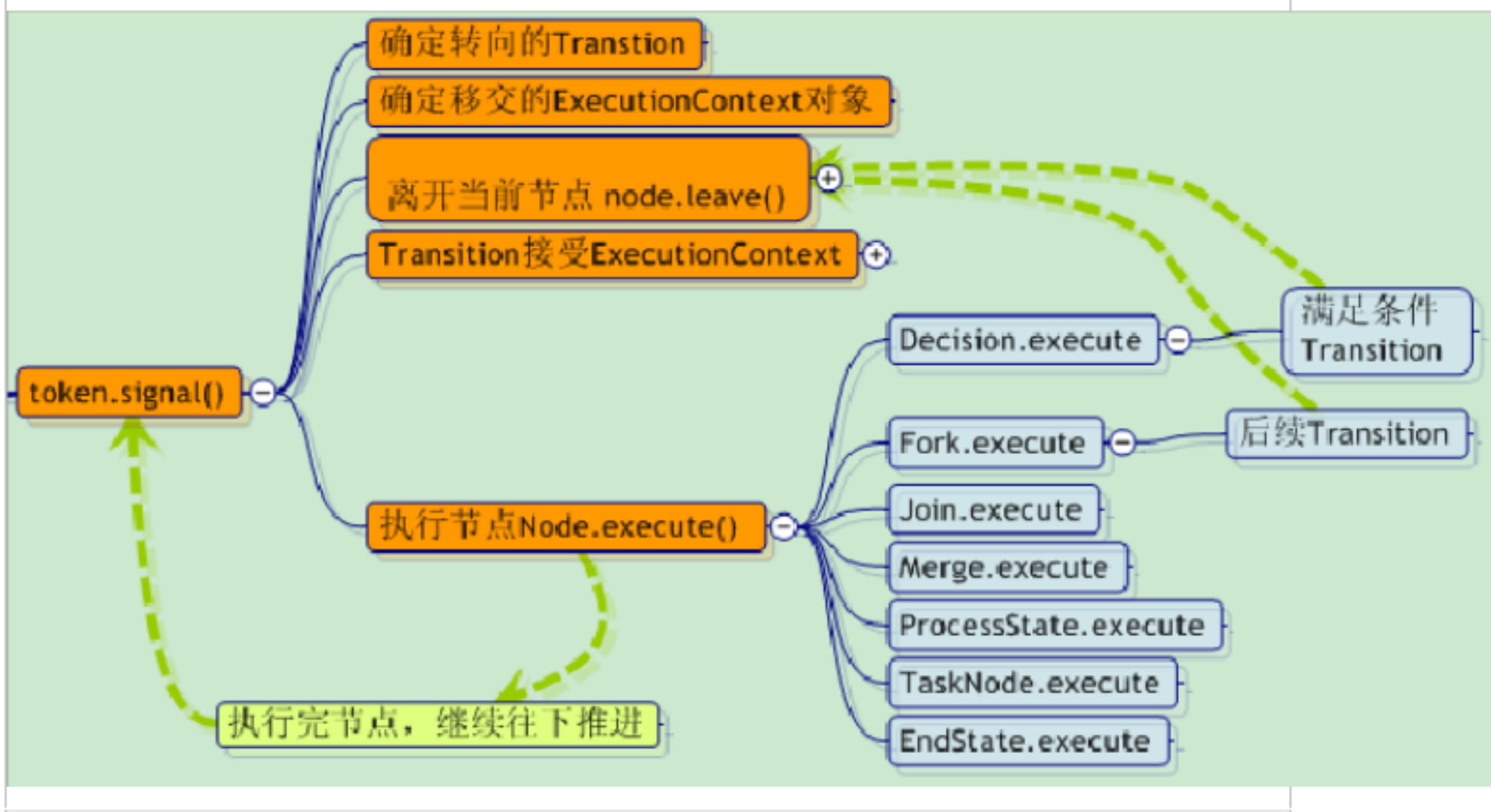
（1） jBpm 没有 NodeInstance 这个概念。利用 Token 和 TaskInstance，jBpm 足以持久化足够的信息，能够让流程实例迅速定位到当前运行的状态。

（2） jBpm 的 Event 已经很丰富，并且这个 Event 是围绕“Token 的转移”而设置的，并不是围绕 Node 的生命周期设置的。

（3） 通常我们需要在 Active 和 Completed 的生命周期内所要操作的分支与聚合，在 jBpm 模型中分别由 Fork、Join 之类的节点替代。所以 jBpm 过分关注 Node 生命周期的管理意义不是非常大。

作为个人，我并不行赏 jBpm 这样抛弃“节点生命周期管理”的实现方式，更行赏 OBE（最早的基于 XPDL 模型的 java 工作流引擎之一）的生命周期约束和管理。但是，也不得不承认，jBpm 规避了“繁琐的状态维护”，反而让处理变得“简易”，也更容易被大家所理解和接受，而这也正是 OBE 逐渐消失的一个原因：过于复杂和臃肿。

让我们在前面那张 jBpm 的“调度机制思维图 ”上，再稍稍补充一点 （为了突出显示，与上图有所改动）。



这张图应该可以很好的诠释出， jBpm 是如何执行各种节点的， 这也是得益于 OO 的“多态与继承 ”特性。

8.2 分支处理

jBpm 的执行机制非常简单，但还是需要稍微补充一下有关 “分支 ”方面的处理。

jBpm 采用 sub token 的机制来解决分支方面的处理： 当遇到有分支的时候， 会为每个分支节点创建一个 child token 。在聚合节点 （Join 或 Merge ），则依赖其同步或异步的聚合方式，来分别处理。

比如我们参看 Fork 节点的执行代码（为了突出重点，省略部分代码）：

```
public void execute(ExecutionContext executionContext) {

Token token = executionContext.getToken();

Iterator iter = transitionNames.iterator();

while (iter.hasNext()) {

String transitionName = (String) iter.next();
```

```
forkedTokens.add(        createForkedToken(token, transitionName)              );

                                    }

                        iter = forkedTokens.iterator();

                        while( iter.hasNext() ) {

                                    //  省略部分代码

                        ExecutionContext childExecutionContext = new

                                    ExecutionContext(childToken);

                        leave(childExecutionContext, leavingTransitionName);

                                    }

                                    }


protected ForkedToken createForkedToken(Token parent, String

                                    transitionName) {

    Token childToken = new Token(parent, getTokenName(parent,

                                    transitionName));

            forkedToken = new ForkedToken(childToken, transitionName);

                                    return forkedToken;

                                    }
```

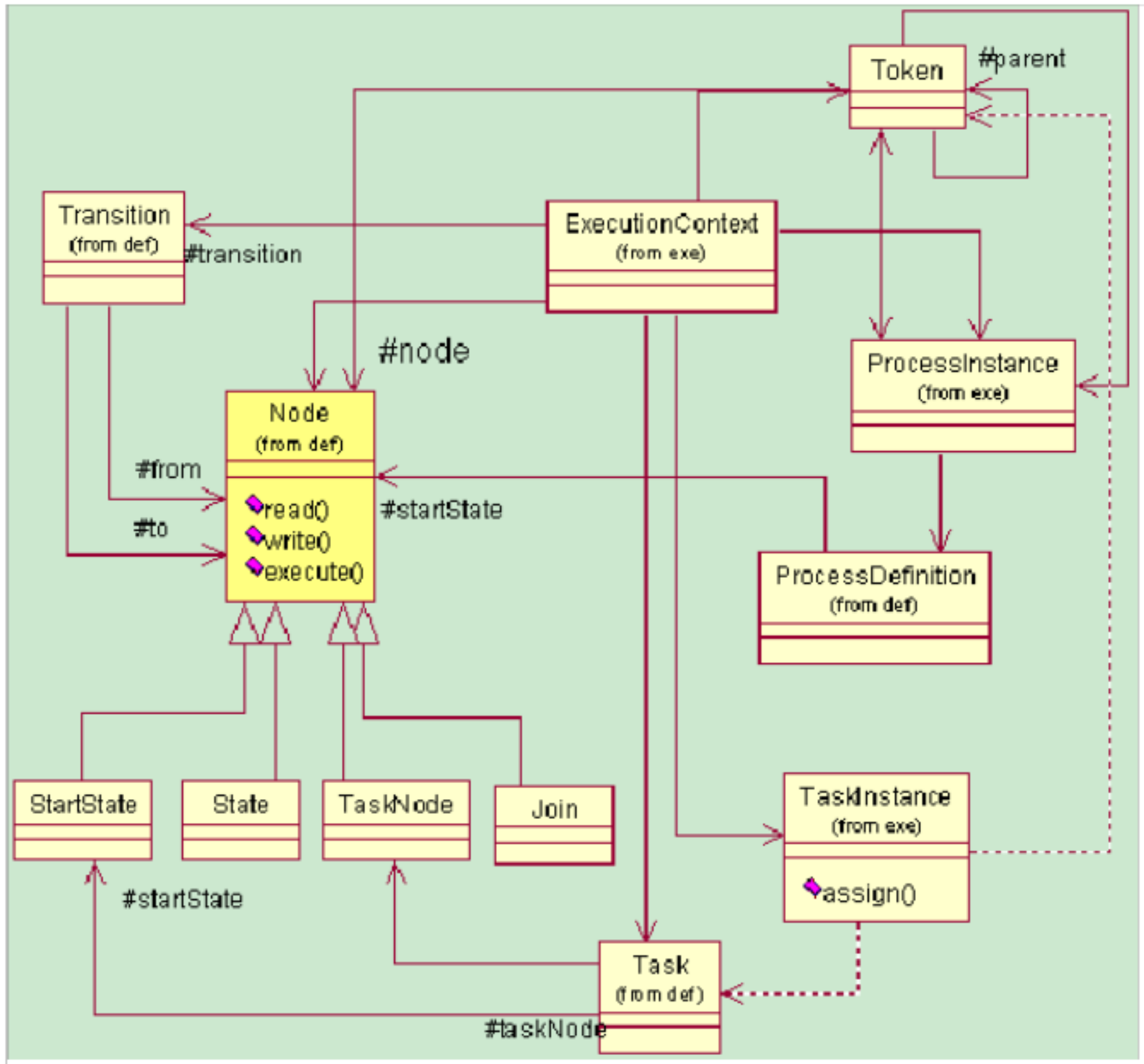
至于 Merge 节点，我想此处不用在累赘的展示，有兴趣的，可以参看 Merge 类的 execute 方法，即可。

9 jBpm 内核结构与实例对象

Jbpm 引擎内核的结构非常 “精简 ”。除了我们上面所说的那些定义对象 （各种 Node 节点和 Transtion ），还有几个与 “运行实例 ”相关的对象。 如下图所示， jbpm 引擎内核对象主要是在 org.jbpm.graph.def 和 org.jbpm.graph.exe 包。

（1） 我们需要描述一个流程实例，所以需要一个 ProcessInstance 对象。

（2） 每个流程实例，都会维护一套属于其自己的 “执行环境 ”，也就是 ExecutionContext 对象。注意，这里是一套，而不是一个。


Token
#parent
Transition
(from def)
#transition
ExecutionContext
(from exe)
#node
ProcessInstance
(from exe)
Node
(from def)
#from
#to
read()
write()
execute()
#startState
ProcessDefinition
(from def)
StartState
State
TaskNode
Join
TaskInstance
(from exe)
assign()
#startState
Task
(from def)
#taskNode